

# A Topology-Based Approach for Lightweight 3-Valued Logic Shape Analysis

Gilad Arnold

June 20, 2005

## Abstract

Although a highly desirable static analysis practice, precise shape analysis is widely considered a computational dead-end for most practical purposes. In this work, we explore ways to overrule this belief, by revisiting and improving over a well-founded shape analysis framework, that is based on 3-valued logic abstraction [6]. While the worst-case exponential asymptotic complexity bound induced by the abstraction domain is not addressed, we propose several practical and theoretical improvements concerned with the representation of the abstract values, as well as the application of abstract transformers, that are aimed at reducing the computational overhead associated with the analysis. We describe a fresh implementation of the analysis framework, which is intently restricted in flexibility and adaptivity compared to the reference framework of [4], but manifests some alternative approaches for implementing abstract elements and transformations, such that are specialized and better suited to achieve a potential gain in performance. Although a highly optimized framework, this implementation proposes a modular architecture that is quite reusable and extendable, where future extensions are considered. Our initial results indicate that the new framework improves by a factor of 20-25 compared to similar analyses performed using the current framework.

## 1 Introduction

The ability to extract the potentially unbounded set of heap configurations that a computer program could induce without running the program, is a highly desirable practice of static program analysis. Its applications range from complete program verification tasks, such as proving the absence of null dereferences, through the assertion of temporal heap properties, such as heap safety properties (e.g., object liveness), and on to practical program optimizations, such as compile-time automatic dis-

covery of program points where explicit object deallocation can be safely added without affecting the correctness of the code. Nonetheless, shape analysis appears to be among the hardest problems in static program analysis: as it is undecidable to prove even simple properties on very small programs manipulating dynamic data structures, with pointers and destructive updates, even the compulsory use of conservative abstraction methods following [3] implies non-trivial frameworks, that in turn induce considerable complexity issues.

In this work, we consider a framework for conducting static shape analysis, that uses logical structures to represent heap topology and shape properties, and applies first-order logic formulas to model the semantics of program transformations [6]. For the purpose of abstraction, (potentially unbounded) sets of (potentially unbounded) logical structures occurring during the analysis of some program, that have similar properties as per the analysis in question, are collapsed into finite, abstract “representative” structures, while keeping these properties intact. Although analyses instantiated by this framework are proven to yield precise and meaningful results compared to the actual (concrete) set of configurations induced by a program [4], it is not so widely studied, let alone deployed in actual production-level compilers or analysis tools: the worst-case complexity of the abstraction is double-exponential by the number of abstraction predicates—namely, by the number of program variables—thus implying a bad “explosion” of the abstract states, requiring a respectively huge number of iterations before a fixed point is reached.

While scalability in the asymptotic sense does not seem to be a feasible goal, given the above complexity concerns, 3-valued logic based shape analysis is commonly considered an exclusively theoretical direction in the research of static analysis: an existing reference implementation in the form of TVLA [4] was used to demonstrate the precision of such analysis, as well as the adaptivity of the 3-valued logic abstraction domain for a wide variety of shape-related problems, but could not effectively scale

```

x = null;
while (...) {
    y = new SLL();
    ...
    y.n = x;  x = y;
}

prev = null;
y = x;
while (y != null) {
    if (...) {
        if (prev == null) x = y.n;
        else prev.n = y.n;
        break;
    }
    prev = y;  t = y.n;  y = t;
}

y = x;
while (y != null) {
    ...
    t = y.n;  y = t;
}

```

Figure 1: A simple Java program that constructs a singly-linked list; traverses its element, possibly deleting some element from the list; and finally traverses the whole list from head to tail.

for programs beyond a few hundreds of lines of code. Figure 1 shows a simple program that constructs a singly-linked list, then traverses it from head to tail, possibly detaching some element from the list based on some criteria. Analyzing the automatically generated intermediate representation of this program with a standard heap shape abstraction in TVLA, yields slightly over 570 abstract structures associated with program locations, and takes over 12 seconds to converge. Given that, it is no wonder that many interesting applications that derive from shape analysis—such as the compile-time memory management described in [2]—can hardly be shown to yield significant practical results, as they cannot be effectively applied to anything beyond small and contrived mini-benchmarks.

This paper describes the major guidelines that led to the design and implementation of a shape analysis framework that is based on the 3-valued logic canonical abstraction of [6]. Concentrating on a fixed, specialized, and highly-optimized abstraction domain, a fixed set of abstraction instrumentation, and a fixed set of basic yet universal transformations, the major contributions of this work seem to be the following.

- A new, fast implementation of the 3-valued logic structure powerset domain, using a graph-based representation of structures, which is aimed at exploiting the sparsity of abstract heap structures, as well as promoting faster operation of both domain-associated operators (i.e., join and meet) and overlying transformer operators (e.g., update formulas, transitive closure computation). This is also a strict implementation of the abstraction domain variant described in [2], assuring precise closure w.r.t. the meet algorithm described in [1, 2].
- A clean—both conceptual and practical—separation between the topology abstraction domain and the semantic transformers abstraction, suggesting new insights about modular architectures of 3-valued structure powerset domain based analysis, and implying greater adaptivity to future framework extension.
- The exclusive use of 3-valued structure powerset domain operators—especially the *meet* operator—for all purposes of refinement and validation of abstract elements. By that, we are able to replace and dismiss two refinement related algorithms inherent to the framework of [6], that consume most of the processing time associated with analyses done by TVLA.

- An empirical improvement by a factor of 20-25 over TVLA, for a restricted set of benchmark programs manipulating recursive data structures.

The rest of this paper is organized as follows. Section 2 introduces 3-valued logic based abstraction for conducting heap shape analysis. Section 3 explores several practical bottlenecks induced by the TVLA framework, and proposes alternative implementation of functionality. Section 4 describes the assumptions, concept, and the inferred guidelines underlying the design and architecture of our implementation, as well as some actual internal mechanisms used to improve the performance of the resulting framework. Initial experimental results are given in Section 5, and Section 6 concludes.

## 2 3-Valued Logic Shape Analysis Overview

We first explain the representation of concrete program heap states and their abstraction, based on the parametric framework of [6]. Then, we describe in greater detail the approach used for representing and implementing abstract transformers underlying that framework.

### 2.1 Abstract domain

#### 2.1.1 Concrete Program States

We represent concrete program states by 2-valued logical structures.

**Definition 1** (Concrete state). A 2-valued logical structure over a vocabulary (set of predicates)  $\mathcal{P}$  is a pair  $S = (U, \iota)$  where  $U$  is the universe of the 2-valued structure, and  $\iota$  is the interpretation function mapping predicates to their truth-value in the structure: for every predicate  $p \in \mathcal{P}$  of arity  $k$ ,  $\iota(p) : U^k \rightarrow \{0, 1\}$ .

In this paper, we assume that the set of predicates includes the binary predicate  $eq$ , and insist that it is interpreted as equality between individuals.

We denote the set of all 2-valued logical structures over a set of predicates  $\mathcal{P}$  by  $2\text{-STRUCT}[\mathcal{P}]$ . In the sequel, we assume that the vocabulary  $\mathcal{P}$  is fixed, and abbreviate  $2\text{-STRUCT}[\mathcal{P}]$  to  $2\text{-STRUCT}$ .

Table 1 shows the predicates used to record properties of individuals for the analysis of the program in Figure 1. We also define additional so-called “instrumentation” predicates to capture properties of individuals such as pointer-aliasing, sharing, cyclicity, and transitive reachability. Instrumentation predicates provide for

more precise information when applying abstraction on a concrete semantics. In particular, in Table 1 we define instrumentation predicates that capture reachability information (via predicates of the form  $r_{x,n}(v)$ ), sharing information (via the predicate  $s_n(v)$ ) and information on cycles in the heap graph (via the predicate  $c_n(v)$ ).

Concrete (2-valued) logical structures are depicted as directed graphs. Each individual of the universe is drawn as a node. A unary predicate  $p(u)$ , which holds for an individual  $u$ , appears next to the corresponding node. If a unary predicate represents a reference variable, then it is shown by having an arrow drawn from its name to the node referenced by the variable. The binary predicate  $n(u_1, u_2)$ , which holds for a pair of individuals  $u_1$  and  $u_2$ , is drawn as a directed edge from  $u_1$  to  $u_2$ , and labeled  $n$ . The predicate  $eq$  is not drawn, since any two nodes are different and every node is equal to itself.

Figure 2(a) shows a concrete program state arising after the execution of the statement  $t = y.n$  at the second loop of the program in Figure 1.

#### 2.1.2 Abstract Program States

The abstract program states we use are based on Kleene 3-valued logic [6], which extends Boolean logic by introducing a third value  $\frac{1}{2}$ , denoting values that may be either 0 or 1. In particular, we utilize the partially ordered set  $\{0, 1, \frac{1}{2}\}$  where  $0 \sqsubseteq \frac{1}{2}$  and  $1 \sqsubseteq \frac{1}{2}$ , with the join operation  $\sqcup$ , defined by  $x \sqcup y = x$  if  $x = y$ , and  $x \sqcup y = \frac{1}{2}$  otherwise.

**Definition 2** (Abstract state). A 3-valued logical structure over a set of predicates  $\mathcal{P}$  is a pair  $S = (U, \iota)$  where  $U$  is the universe of the 3-valued structure, and  $\iota$  is the interpretation function mapping predicates to their truth-value in the structure: for every predicate  $p \in \mathcal{P}$  of arity  $k$ ,  $\iota(p) : U^k \rightarrow \{0, 1, \frac{1}{2}\}$ .

An abstract state may include *summary nodes*, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state. A summary node  $u$  has  $eq(u, u) = \frac{1}{2}$ , indicating that it may represent more than a single individual.

Abstract (3-valued) logical structures are also depicted as directed graphs, where unary predicates denoting reference variables, as well as binary predicates, with  $\frac{1}{2}$  values are shown as dotted edges. Summary individuals appear as double-circled nodes.

We denote the set of all 3-valued logical structures over a set of predicates  $\mathcal{P}$  by  $3\text{-STRUCT}[\mathcal{P}]$ , usually abbreviating it to  $3\text{-STRUCT}$ .

We define a partial order on structures, denoted by  $\sqsubseteq$ , based on the concept of *embedding*.

Predicates	Intended Meaning
$eq(v_1, v_2)$	Is $v_1$ equal to $v_2$ ?
$\{x(v) : x \in PVar\}$	Does reference variable $x$ point to object $v$ ?
$n(v_1, v_2)$	Does the $n$ field of object $v_1$ point to object $v_2$ ?
$\{r_{x,n}(v) : x \in PVar\}$	Is $v$ reachable from reference variable $x$ along a sequence of $n$ fields?
$s_n(v)$	Do two or more $n$ fields of heap elements point to $v$ ?
$c_n(v)$	Is $v$ on a directed cycle of $n$ fields?

Table 1: Predicates used for shape analysis of the program in Figure 1, and their intended meaning. The set of pointer variables in a program is denoted by  $PVar$

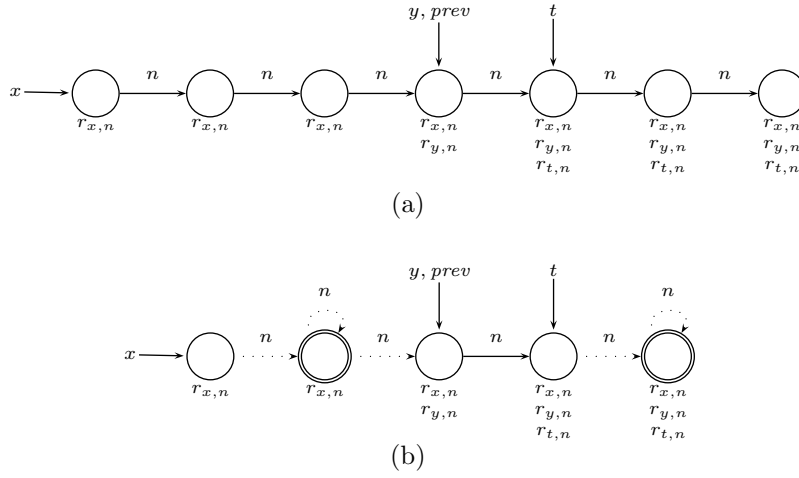


Figure 2: (a) A concrete program state arising after the execution of the statement  $t = y.n$ ; (b) An abstract program state approximating the concrete state in (a)

**Definition 3** (Embedding). Let  $S = (U, \iota)$  and  $S' = (U', \iota')$  be two structures and let  $f : U \rightarrow U'$  be a surjective function. We say that  $f$  *embeds*  $S$  in  $S'$ , denoted  $S \sqsubseteq^f S'$ , if for every predicate  $p \in \mathcal{P}^{(k)}$  and  $k$  individuals  $u_1, \dots, u_k \in U$ ,

$$p^S(u_1, \dots, u_k) \sqsubseteq p^{S'}(f(u_1), \dots, f(u_k)) . \quad (1)$$

We say that  $S$  is *embedded* in  $S'$ , denoted  $S \sqsubseteq S'$ , if there exists a function  $f$  such that  $S \sqsubseteq^f S'$ . We also say that  $S'$  *approximates*  $S$ .

The embedding order is used to define a concretization function for a single 3-valued structure  $S$  by  $\sigma(S) = \{S' \in 2\text{-STRUCT} \mid S' \sqsubseteq S\}$ . The concretization of a set of 3-valued structures is defined by  $\gamma(XS) = \bigcup_{S \in XS} \sigma(S)$ .

The embedding order induces a Hoare preorder on sets of 3-valued structures.

**Definition 4** (Powerset order). For sets of structures  $XS_1, XS_2 \subseteq 3\text{-STRUCT}$ ,  $XS_1 \sqsubseteq XS_2$  if and only if  $\forall S_1 \in XS_1 : \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2$ .

We are now ready to present the general abstract domain used in our abstract interpretation framework.

**Definition 5** (Core abstract domain). The abstract domain  $D_{3\text{-STRUCT}}$  consists of all finite sets of 3-valued structures that do not contain non-maximal structures,  $\{XS \subseteq 3\text{-STRUCT} \mid \forall S_1, S_2 \in XS : S_1 \sqsubseteq S_2 \implies S_1 = S_2\}$ , partially ordered as in Definition 3.<sup>1</sup>

Together with the obvious definition of a bottom element (the empty set), a top element (a set containing one empty structure, and one structure that contains a single summary node and whose predicates are all interpreted as  $\frac{1}{2}$ , for every node-tuple assignment), join operator (set union minus non-maximal structures), and meet operator (defined via join),  $D_{3\text{-STRUCT}}$  forms a lattice.

### 2.1.3 Bounded Program States

Note that the size of a 3-valued structure is potentially unbounded and that 3-STRUCT is infinite. This also implies that sets of such unbounded structures could potentially contain an infinite number of elements. The analyses studied in [6], one of which is implemented in §4, rely on a fundamental abstraction function for converting a potentially unbounded structure—either 2-valued or 3-valued—into a bounded 3-valued structure.

A 3-valued structure is said to be *bounded* if for every two distinct individuals in its universe there exists a unary predicate  $p$  such that either  $p^{S_1}(u_1) = 0$  and

<sup>1</sup>Disallowing non-maximal structures ensures a partial order on the sets.

$p^{S_2}(u_2) = 1$  or  $p^{S_1}(u_1) = 1$  and  $p^{S_2}(u_2) = 0$ .<sup>2</sup> We denote the set of all bounded 3-valued structures over a set of predicates  $\mathcal{P}$  by  $B\text{-STRUCT}[\mathcal{P}]$ . The abstract domain  $D_{B\text{-STRUCT}}$  is a sub-lattice of  $D_{3\text{-STRUCT}}$ , containing all (finite) sets of bounded structures that do not contain non-maximal structures.

The abstraction function  $\beta_{blur}^{\mathcal{P}} : 2\text{-STRUCT}[\mathcal{P}] \rightarrow B\text{-STRUCT}[\mathcal{P}]$  converts a (potentially unbounded) 2-valued structure into a bounded 3-valued structure, by merging all individuals with the same values for all unary predicates. Namely,  $\beta_{blur}^{\mathcal{P}}((U, \iota)) = (U', \iota')$ , where  $U'$  is the set of equivalence classes in  $U$  of nodes with same values for all unary predicates, and the interpretation  $\iota'$  of each predicate  $p \in \mathcal{P}^{(k)}$  and each  $k$  individuals  $c_1, \dots, c_k \in U'$ , is given by

$$p^{S'}(c_1, \dots, c_k) = \bigsqcup_{u_i \in c_i} p^S(u_1, \dots, u_k) .$$

Figure 2(b) shows a bounded structure obtained from the structure in Figure 2(a).

The abstraction function  $\beta_{blur}$  is called *canonical abstraction*, and serves as the basis for the abstract interpretation described in [6]. In particular, it serves as the basis for defining various different abstractions for the (potentially unbounded) set of 2-valued logical structures that may arise at a program point. We also define the function  $\alpha$ , which extends  $\beta_{blur}$  for sets of structures by  $\alpha(XS) = \bigsqcup \{\beta_{blur}(S) \mid S \in XS\}$ .<sup>3</sup> Along with a respective concretization function  $\gamma$ , that is defined by means of  $\alpha$ , the concrete domain 2-STRUCT and the abstract domain 3-STRUCT form a Galois connection, and hence comply with the framework requirements of [3].

## 2.2 Abstract transformers

The abstract interpretation framework described in [6] models the semantics of program transformations w.r.t. state predicates by first order logic formulas with transitive closure over the interpretation associated with each inbound structure. For example, the transformation induced by the program statement  $t = y.n$  from the example in Figure 1, is modeled by

$$\begin{aligned} t(v) &\leftarrow \exists v' : y(v') \wedge n(v', v) \\ r_{t,n}(v) &\leftarrow r_{y,n}(v) \wedge (c_n(v) \vee \neg y(v)) . \end{aligned}$$

While the correctness of this transformer is evident when it is applied to some concrete structure, it is not at

<sup>2</sup>The notion of a bounded structure can be generalized by considering any subset of the set of unary predicates to serve as *abstraction predicates*, as done in TVLA.

<sup>3</sup>The operator  $\bigsqcup$  is the least upper bound on the lattice  $D_{B\text{-STRUCT}}$ .

all obvious that the same holds for the case of an abstract structure, such whose predicates may evaluate to  $\frac{1}{2}$ . Nonetheless, the *embedding theorem* proved in [6] shows that the result of such a transformation is always a *sound approximation* of the best transformer.<sup>4</sup>

Although evaluating update formulas over abstract state elements in a straightforward manner is indeed sound, it would cause a significant loss of precision upon each transformation, thus leading to a useless analysis. In order to address this problem, and since a best transformer is generally intractable in this case,<sup>5</sup> two auxiliary algorithms are described in [6] in order to apply *partial concretization* of abstract structures.

**Focus.** Given a 3-valued structure  $s$  and some first order formula  $\phi$ , this algorithm essentially enumerates the set of structure that is both embedded in  $\{s\}$ , and whose corresponding concrete range is equal to that of  $\{s\}$ , and for which  $F$  evaluates to either 0 or 1, for each structure.

Figure 3(a) shows a canonically bounded doubly-linked list structure, that is processed by the transformer associated with the statement  $\mathbf{t} = \mathbf{y.n}$ . Figure 3(b) shows the result of the focus operator on this structure, given a focus formula

$$y(v) \wedge n(v, v') ,$$

which is applied to each pair of individuals  $v, v'$  in the structure operand. As it can be seen, the evaluation of the focus formula on each resulting structure yields a deterministic value. Note, however, that the first of the resulting structures does not satisfy the integrity constraints, as the suffix of the list appears to be disconnected, and the other two structures are not as precise as could be, as several binary predicates (i.e., list pointers) are interpreted as  $\frac{1}{2}$  where they can safely be tightened to either 0 or 1.

**Coerce.** Given a 3-valued structure, the functionality of this operator is two-fold: by exhaustive evaluating of formulas derived from structure integrity rules, over all tuples of nodes, it both dismisses structures for which some constraint is breached (e.g., the first structure in Figure 3(b) does not satisfy the reachability property that is expressed by the  $r_{x,n}$  predicate for the summary node), but also tightens predicate values where such a tightening is implied by the

<sup>4</sup>Note, that the notion of a “best transformer” in the case of 3-valued abstract structures implies an instrumented concretization function, such that assures the validity of concrete structures w.r.t. the *integrity constraints* expressed by instrumentation predicates.

<sup>5</sup>Recent work has been done in [7] to elevate the applicability of best transformers for the case of 3-valued analysis, using a theorem prover to evaluate first order logic formulas.

constraints (e.g., the second structure of Figure 3(b), in which the back pointer in the list is tightened to 1, and the self loop edges are tightened to 0).

The result of coercion for the focused structures is shown in Figure 3(c).

The details involving the algorithms used to compute both the above operators are found in [6], and are omitted here for brevity.

In the following section we discuss several issues having to do with the implementation of the abstract domain, its associated operators, and the various stages of the abstract transformers, as they are evident in the reference implementation of TVLA [4].

### 3 Issues with 3-Valued Shape Analysis Implementation

Any implementation effort compliant to the specification of the 3-valued analysis framework described in §2 would obviously result in a particularly complicated piece of code. Furthermore, constructing such a software analysis tool that can be parameterized by customizable analysis specification—predicates, abstraction instrumentation and integrity rules, and abstract transformers expressed as first order logic formulas—in a relatively short time, and without a previous reference or practical experience with the practical implications of the implemented abstraction, is a difficult task.

Although a pretty mature software package by this time, the TVLA framework [4] was developed in such a way, and so it is not surprising that applying it to analyze actual programs normally involves significant time and space overheads, even when the program in question is very small, often no more than a few dozens of lines. Analyzing the example Java program in Figure 1, translated by the J2TVLA front-end into a simplified intermediate representation that induces a CFG of 50 nodes (program locations) and a similar number of edges (transformations), with a tweaked version of TVLA running a standard heap abstraction, would take over 12 seconds, consume over 3 MB of memory, and converge to a steady state with a little more than 570 structures. As such a penalty for that small of an input cannot be justified, these symptoms make it very hard to convince the program analysis community with the importance and usefulness of 3-valued shape analysis.

With that in mind, we briefly explain some of the causes for performance bottlenecks that are associated with TVLA analyses.

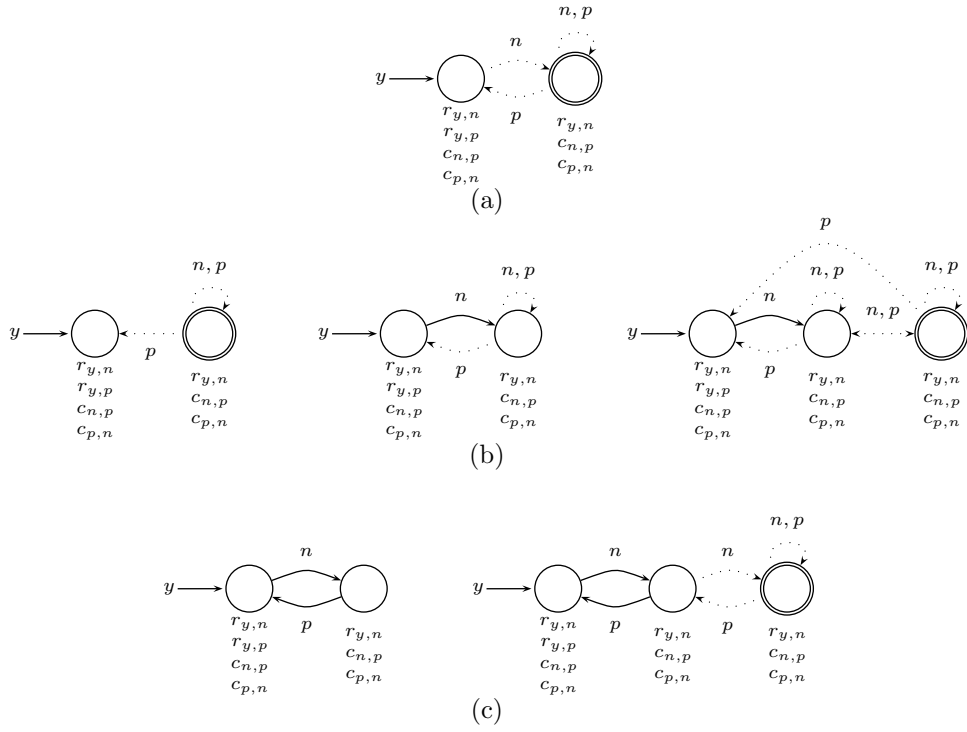


Figure 3: The stages of an abstract 3-valued structure refinement: (a) the original inbound structure; (b) the structures resulting after applying the focus operator with the formula  $y(v) \wedge n(v, v')$ ; (c) the remaining structures after integrity validation and predicate interpretations tightening with the coerce operator

**Inadequate data representation.** The generality and intended flexibility of TVLA—in particular, the fact that it supports predicates of arbitrary arity—sometimes lead to representation that is inappropriate for efficient processing of 3-valued structures. One example to that is the representation of predicate interpretation values, by having each predicate use a hash table, in which tuples of nodes are mapped to Kleene values. While this is aimed at promoting fast predicate evaluation, it has some bad effects due to the fact that traversing transitive binary connections (an operation that’s heavily used within the abstract transformers) requires exhaustive iterations over those interpretation tables. Another impact is a significant slowdown incurred by *canonical name* extraction, an operation whose inefficiency badly affects the performance of several operators, such as the algorithm that’s used to compute meet.

**Non-optimized domain operators.** As mentioned, one example for that is the construction of canonical names, which is often required for structure embedding / isomorphism verification, as well as a key operation when computing meet using the algorithm described in [1]. Although a seemingly trivial part in the algorithm, this phase happens to consume most of the time that is spent by the meet operator.

**Costly refinement algorithms.** The focus and co-erce algorithms—the latter in particular—described in §2.2, induce a significant performance overhead, as it is evident through TVLA analysis statistics. That is due to exhaustive constraint formula evaluation that has to be carried out upon each update operation.

**Naive evaluation of updates.** An example for that is the inefficient evaluation of existential update formulas, as well as transitive closure formulas, during transformation evaluation. Obviously, properties whose evaluation includes a transitive closure formula, such as reachability by a sequence of field references, correspond to graph reachability properties, and hence their values among different nodes are not independent. In particular, if  $r_{x,n}$  holds for some node  $v$ , implying a path of  $n$  field references from some node  $v'$  which is pointed by  $x$ , to  $v$ , then the fact that  $n(v, v'')$  holds for some  $v''$  immediately implies that  $r_{x,n}(v'')$  holds. Thus, there seems to be an obvious analogy between transitive closure evaluation and global graph reachability computation,

but this correspondence is neglected in favor of a naive and sub-optimal processing.

In the following section we describe an alternative implementation of the 3-valued shape analysis framework, that addresses the above drawbacks in the reference implementation, and thus is likely to improve the performance of the analysis significantly.

## 4 An Alternative Prototype Implementation

We describe the major characteristics of an alternative implementation of a restricted version of the 3-valued domain analysis framework.

### 4.1 Goals, limitations, and assumptions

At the bottom line, we want to be able to run standard 3-valued shape analysis with performance characteristics that are an order of magnitude better than those of TVLA. We hope that, by introducing a “skim” and specialized implementation of the 3-valued abstraction domain and abstract transformers, we would allow such an analysis to scale slightly better, and thus make it more accessible for purposes of practical research. In this work, we do not intend to improve on asymptotic worst-case bounds induced by the abstraction domain itself, and we consider that a separate topic for future study.

Our proposed implementation is restricted in several manners, compared to the generic framework of TVLA.

**Fixed predicate arity.** As opposed to TVLA, in which predicates in the abstraction can be defined to be of any arity  $k \geq 0$ , we restrict our framework to only support nullary, unary, and binary predicates. This is guided by the fact that the heap relations modeled by shape analysis always correspond to either of those types of predicates: method-local reference variables are normally modeled by unary predicates,<sup>6</sup> and reference fields are modeled by binary predicates. We also want to model Boolean variables and fields, in order to increase the precision of our abstract interpreter: local Boolean variables are modeled by nullary predicates, and Boolean object fields are modeled by unary predicates. In the

<sup>6</sup>In this context, it is worth mentioning that several approaches for interprocedural shape analysis extend this notion, by modeling local references as binary relations between “stack objects” (frames) and heap objects.

case of our specialized abstraction, all instrumentation properties are modeled by unary predicates (see below).

**Fixed abstraction instrumentation.** Our proposed implementation supports a predefined set of instrumentation predicates, that can be either instantiated or not during the initialization of the analysis. The supported instrumentation includes the following.

**Reachability.** For each pair consisting of a (unary) predicate  $x$ , representing a reference variable, and a (binary) predicate  $f$ , representing a reference field, a (unary) instrumentation predicate  $r_{x,f}$  can be generated.  $r_{x,f}(v)$  holds for some node  $v$ , iff that node is reachable from a reference variable  $x$  through a sequence of (zero or more)  $f$  dereferences. Note, that the computation of such a predicate value might require the explicit evaluation of reachability transitive closure, which we will discuss later.

**Cyclicity.** For a (binary) predicate  $f$ , representing a reference field, a (unary) instrumentation predicate  $c_f$  can be generated.  $c_f(v)$  holds for some node  $v$ , iff that node resides on a cycle formed by a sequence of (one or more)  $f$  references.

**Sharing.** For a (binary) predicate  $f$ , representing a reference field, a (unary) instrumentation predicate  $s_f$  can be generated.  $s_f(v)$  holds iff  $v$  is pointed from more than one object by an  $f$  field.

**Cancel.** For a pair of (binary) predicates  $f, f'$  representing reference fields, a (unary) instrumentation predicate  $c_{f,f'}$  can be generated.  $c_{f,f'}(v)$  holds iff for any node  $v'$  such that  $f(v, v')$  holds,  $f'(v', v)$  must hold.

The above set of supported instrumentation is considered to be “standard” for general purpose shape analysis tasks. While more complicated instrumentation properties may lead to greater precision—binary reachability instrumentation is one example—we believe that their induced computational overhead is not worth this gain in precision.

Although in this proposed framework we do not implement fully automatic inference of instrumentation values—commonly referred to as *differencing* [5]—the fact that we implement a fixed set of transformers implies that both soundness and precision of their value assignment is controlled by the

framework itself, and implemented within the fixed set of provided transformers (see below).

**Fixed set of hard-coded transformers.** In a similar manner to the above, our proposed framework supports a fixed set of transformers, which we believe to be universal w.r.t. modeling the semantics of common imperative languages with pointers and destructive updates. Specifically, we support empty statements (skip); assignment of either a null value, a reference variable’s value, or a reference field value, to either a reference variable or field; conditional (non-) equivalence of two reference variables or a reference variable and a null value; a similar set of statements for the case of Boolean variables, which are also modeled by the abstraction; object allocation and deallocation statements; and procedure call/return statements. We assume the use of a simplifying front-end to yield a canonical intermediate form that complies with this set of transformations.

Note, that at this point we do not support array involving operations, and leave this for future development. Also note that our current support for procedure call/return is completely schematic, as context-sensitivity is not embedded in our framework, a topic that we consider to be future work as well.

Our proposed set of transformers is to be implemented in code, as part of the framework implementation. As it is derived from a well-known TVLA analysis specification, it is assumed to be sound and quite precise w.r.t. the available abstraction instrumentation. The details of the actual transformer semantics are omitted for brevity.

Note, however, that the above work assumptions do not limit the generality of our abstract domain implementation, other than the loosening implied by the bound on predicate arity. This actual implementation is further described in §4.3.

## 4.2 Architecture outline

Figure 4 shows a general block diagram explaining the architecture of our framework. Blocks with related or dependent functionality have the same color. Thus, it can be seen that our implementation of the 3-valued abstraction domain, along with that of the canonical abstraction, are independent from any particular analysis layout. The latter is defined using the analysis-dependent modules, that include the schema (responsible for predicate

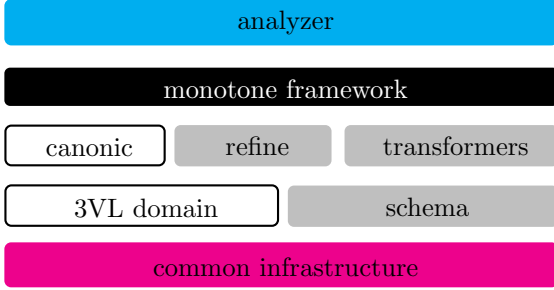


Figure 4: The general architecture of the framework implementation

definition and layout, construction of analysis-specific instrumentation scheme, etc.), the refinement module (responsible for constructing and applying refinement operators), and the abstract transformers module that uses it. A monotone framework front-end provides a unified singleton analysis interface. A simple analyzer, capable of parsing TVLA-like specification, constructing an analysis, and conducting the chaotic iterations algorithm, is provided for evaluation purposes.

The above diagram suggests that further extensions to the analysis engine, like support for currently unsupported abstraction or instrumentation features, can be undertaken without needing to modify the abstract domain implementation.

### 4.3 3-valued structure powerset domain implementation

We briefly sketch the outlines of our topology-based approach for representing and manipulating 3-valued structures and powerset elements.

#### 4.3.1 Representing 3-valued structures

In our implementation, 3-valued structures are represented by directed graphs with attributes associated to graph nodes and edges, corresponding to predicate interpretations involving these (pairs of) nodes: node-attached attributes correspond to unary predicates, while edge-attached attributes correspond to binary predicates. A structure also stores the interpretation of nullary predicates’ values.

The use of a graph-based representation has several intuitive justifications. Initially, it is a concise representation that exploits the sparsity which is common to actual (and also to abstract) heap configurations: as it is observable from the transformations’ code, most up-

dates only care for non-false predicates, hence a sparse data structure is likely to be an advantage. But, furthermore, we believe that its greatest advantage lies in the fact that—unlike the TVLA node-tuple hash-based representation—the preservation of actual topology promotes a faster evaluation of transformer updates due to the ability to easily decode “heap path” relation sequences. Consider, for example, the following formula used for updating the value of a reference variable predicate when applying the transformer corresponding to  $\mathbf{t} = \mathbf{y.n}$ ,

$$t(v) \leftarrow \exists v' : y(v') \wedge n(v', v) .$$

While it’s naive evaluation—which is inherent to the hash-based predicate interpretation representation—would have to consider every possible pair of nodes, a topology-aware representation would only consider, for each node  $v'$  that is referred to by  $y$ , the subset of nodes  $\{v \in U\}$  to which a non-false value  $n(v', v)$  (i.e., a graph edge with non-false  $n$  attribute) exists. Since our topology is likely to be rather sparse, the resulting evaluation process is expected to be more economical than a full-scale naive quantification. Put in a general way, it is expected to promote faster computation wherever an existential formula is used to infer non-false binary connectivity, and there are quite a few cases of that in the shape analysis transformations specification.

Our topology-based representation is also likely to minimize overhead when transitive closure of binary relations needs to be computed. Consider the following formula, for example, that is used for updating the value of reachability instrumentation predicates upon an assignment statement  $\mathbf{t} = \mathbf{y.n}$ ,

$$r_{t,f}(v) \leftarrow \begin{cases} r_{y,f} \vee (\exists v' : y(v') \wedge n(v', v)) \\ \text{if } \exists v', v'' : y(v') \wedge c_{n,f}(v') \wedge n(v', v''), \\ \exists v', v'' : y(v') \wedge n(v', v'') \wedge f^*(v'', v) \\ \text{otherwise,} \end{cases}$$

evaluated for each reference field  $f \neq n$  for which  $r_{y,f}$  is defined. (Note, that due to the Kleene way of logical value interpretation, it might be the case that both sub-formulas need to be evaluated.) Here, the notation  $f^*$  stands for a transitive conjunctive closure of  $f$  predicate pairs sequence: while the naive way of evaluation—which is, again, inherent to a topology-unaware representation—would have to consider all the nodes on each iteration of the transitive expansion, our implementation can compute a complete closure in strictly linear time.<sup>7</sup>

<sup>7</sup>Moreover, in the case of the above formula, our specialized transformer implementation combines both the efficient existential

### 4.3.2 Implementing lattice operators

Our framework implements the join and meet operators, as well as the embedding relation ( $\sqsubseteq$ ), based on a fast heuristic approach for computing structure embedding and correspondence relations, as described in [1, 2]. For brevity, we omit the details and only mention some of its implications.

While the matching procedure underlying the enumeration of either embedding or correspondence relations is quite efficient, our experience hints that the operators' implementation spends most of its time in constructing matching graphs, validating matching results and constructing intermediate structures (in the case of meet). The former implies that an efficient representation of node attributes (unary predicates) that supports fast Kleene operators over the whole vector of attributes, is highly desirable. The other two strengthen this intuition, however, they also hint that a topology-aware representation of binary predicate interpretation may promote the validation and generation of intermediate structure. Hence, we expect our implementation to improve in that essence as well.

Our implementation of the powerset domain complies with the strict 3-valued structure powerset lattice that is discussed in [2]. This variant has some interesting properties, such as closure under a unique greatest lower bound, and the unification of the meet operator with that of the general (containing) 3-valued structure powerset domain. This guarantees that meet operations using the algorithm described in [1, 2] would yield precise values w.r.t. the bounded structure abstract domain that we are using for the analysis.

## 4.4 Analysis specification

The proposed framework uses a unique approach to efficiently represent the analysis specification, as well as applies a new method for achieving precise abstraction refinement without the use of the dedicated, general-purpose symbolic refinement algorithms mentioned in §2.2.

### 4.4.1 Abstraction schema

We provide an infrastructure for constructing per-program analysis, by means of adding predicates that correspond to program-specific variables and fields. While this for itself is trivial, we do use a topological

---

quantifier inference, and the fast transitive closure computation, presumably yielding a rather economical evaluation process compared to the exhaustive one.

representation (a graph) to represent the relations between those predicates, and their induced instrumentation. For example, assuming a schema which contains a single unary predicate  $x$  and two binary predicates  $n_1$  and  $n_2$ , we represent their interrelated instrumentation predicates so that they can be effectively retrieved and traversed when applying transformation updates. Thus, an instrumentation predicate  $r_{x,n_1}$  is attached to the directed edge  $(x, n_1)$ , instrumentation predicates  $c_{n_1,n_2}$  and  $c_{n_2,n_1}$  are attached to directed edges  $(n_1, n_2)$  and  $(n_2, n_1)$ , respectively, and instrumentation predicates  $c_{n_1}$  and  $s_{n_2}$  are attached to nodes  $n_1$  and  $n_2$ , respectively. This layout promotes faster access to predicates that are relevant to an update transformation, as in the case of the update formula for  $r_{x,f}$  shown in §4.3.1, applied to each  $f \neq n$  for which  $r_{x,f}$  is defined.<sup>8</sup>

### 4.4.2 Domain operators based refinement

An interesting approach, which is a novelty of our implementation, is used to perform abstraction refinement, needed in order to retain precision of the abstract states, as explained in §2.2. Our approach is based on the exclusive use of domain-inherent operators, that is the meet and join operators. By doing so, we are able to avoid the use of focus and coerce, both of which are problematic parts of the 3-valued analysis framework of [6].

While the principal use of meet to express structure refinement (i.e., focus) and filtering (i.e., precondition enforcement) is explained and exemplified in [1], the applicability of such a mechanism for arbitrary refinement needs is in question: as it is well-known that 3-valued structures are equivalent to a restricted, decidable family of logical formulas, it is evident that it cannot be used to express *any* refinement formula that the general focus algorithm can handle, as the latter evaluates first-order logic formulas which are generally undecidable. Nonetheless, one of the important findings implied by our implementation, is that a simple meet-based refinement is sufficient for the refinements needs of the standard transformation set that the framework supports. Specifically, we found that the only refinement required through the standard analysis is such that focuses a reference by some reference variable, and such that focuses a reference of the form  $y.n$ , where  $y$  is a reference variable and  $n$  is a reference field.

However, an even more surprising fact that we manage to retain the precision of our transformations even without the integrity constraint validation, as well as the

---

<sup>8</sup>This could be exemplified by a diagram, but was not due to time constraints. . .

tightening functionality of coerce. These two criteria can be explained separately.

**Constraint validation.** In the case of an arbitrary focus formula, a resulting “focused” structure might not (conservatively) satisfy the refinement constraints, as expressed by the instrumentation predicates. An example for such a case is the first structure shown in Figure 3(b): the fact that  $r_{y,n}$  evaluates to 1, while the value of  $n$  from the node for which  $y$  holds and the summary node evaluates to 0, causes the constraint to be breached.

However, in the case of a tailored structure-based refinement, we can obviously avoid this kind of refinement consequences, if we assure that every structure used for refinement—that is, to which a meet operator is applied together with some currently analyzed structure—satisfies the integrity constraints induced by instrumentation predicate implementation. An explicit example for a refinement structure set that complies with this requirement is given in [1].

**Abstraction tightening.** While the refinement induced by a structure as the one described above normally results in a set of structures that are “sane”, it does not necessarily yield the most precise structures that it could have yielded, judging by the interpretation of their instrumentation predicates. An example for such a resulting structure is the second structure shown in Figure 3(b). The sources for imprecision with this structure lie within the fact that the  $p$  binary predicate from the right-hand side node to the left-hand side node evaluates to  $\frac{1}{2}$ , whereas by the fact that  $c_{n,p}$  holds for the left-hand side node it could have been tightened to 1; and the fact the both  $n$  and  $p$  binary predicates for the loop case of the right-hand side node evaluate to  $\frac{1}{2}$ , whereas by the fact that  $c_n$  and  $c_p$  do not hold for that node they could be tightened to 0. We show how we resolve the former case, and claim (without explanation) that the same method can be applied to the latter case.

A straw-man solution to the tightening problem can be found in the form of sub-case partial concretization of the refining structure set, down to the point where all cases of a back edge  $p$  from the right-hand side node to the left-hand side one are enumerated deterministically (i.e., to 0 and 1 values). While this doesn’t seem to be too complicated for the case of just two confluent reference fields  $n$  and  $p$ , this is definitely problematic in the presence of an arbitrary number of such fields, namely  $n_1, \dots, n_k$ , as

it would require an enumeration in the order of  $2^k$  structures to cover all sub-cases. Furthermore, all that mass is there only to comply with the one particular case that the refined structure corresponds to...

Instead, we turn to a staged method for applying refinement: we split the various cases of the value of the  $n$  field into distinct elements. Then, for each such element, we construct a set of refinement sub-cases, each of which focuses (among other things) on a particular instrumentation predicate value  $c_{n,n_i}$ , with  $1 \leq i \leq k$ , while keeping all other predicates  $c_{n,n_j}$ , and respectively returning  $n_j$  edges, non-deterministic (i.e.,  $\frac{1}{2}$ ), for all  $j \neq i$ . We know that each resulting abstraction element generated by meeting the processed structure with such a sub-case refinement element must have the relevant  $n_i$  back edge focused. Thus, we now generate the meet of all non-empty resulted elements, to get a single result in which all these back edges are focused. Finally, we join the result of all these (major) cases, to get the complete set of refined structures.<sup>9</sup>

In the following section we describe preliminary results of shape analysis run by our framework, and compare them to those of TVLA.

## 5 Initial Experimental Results

We have tested our framework implementation for a set of mini-benchmark programs, with whose analysis by TVLA we are already acquainted. While these benchmarks were far from thorough, and as they were mostly used for tuning the implementation of transformations and refinement operators, rather than testing the performance of our analysis tool, we only describe the results for the test program that is shown in Figure 1. This benchmark induced the heaviest performance penalty compared to its miniature size, as the deletion of a list element implies a relatively large number of possible heap configurations, corresponding to the different possible positions of the deleted element within the list.

We used a slightly tweaked version of TVLA to analyze this program. Our analysis was using the standard shape abstraction—the equivalent of what we have implemented in our framework—with manual instrumentation update (i.e., without using automatic instrumentation differencing, which tends to slow the analysis down

<sup>9</sup>I’d love to give a detailed example by figure, but this was not possible due to the combination of complexity of such a construct, and the time constraints on the authoring... this method can be shown to work empirically via the implementation itself.

by an approximate factor of 2). The result is an analysis that takes more than 12 seconds to run, and consumes over 3 MB of memory (peak), yielding slightly over 570 structures for a set of 50 CFG nodes.

When run with our framework implementation, the analysis was completed in less than 500 milliseconds, yielding the same number of structures. Note that, as our framework is implemented in C, this time includes the deallocation of all intermediate data objects that are allocated during the analysis. Unfortunately, we did not yet implement dynamic memory usage statistics, so we cannot provide any evidence for memory footprint improvement, at the moment. Since the above tests were taken on a 1.6 GHz laptop machine, with the framework code compiled and run with a Cygwin emulator, we also conducted another test on a stronger, 2.4 GHz dual-processor machine running Linux. The time penalty in this case dropped below 150 milliseconds for the whole analysis.

While these results are not that astonishing, given the size of the analyzed program and with comparison to real-world static analyses, they do indicate a performance improvement by a factor of 20-25 over that of TVLA. An additional interesting detail about this result is that most of the analysis time seems to be spent on the execution of join operators, and in particular those that are used to propagate updated structures between CFG nodes. While this is a surprising result, it is also an encouraging one, as it both indicates that our approach for conducting structure refinement and update is indeed highly effective, as well as suggesting that further significant improvement can be achieved by optimizing a small and restricted part of the implementation (namely, the join operator). Such an optimization is left for future work.

In addition to that, we have performed several other tests on programs whose induced analysis scheme involves cancel instrumentation predicates, in the presence of which it was shown that more complicated refinement methods are required. Our empirical results indicate that our refinement method sketched in §4.4.2 indeed results in precise and well-tightened structures.

## 6 Conclusion

We have described our implementation of the 3-valued logic based shape analysis, that is capable of conducting detailed and precise analyses that are comparable to those performed using TVLA. We explained our unique approach for implementing this framework, which includes a specialized set of data structures used to repre-

sent 3-valued structures and analysis schemas, and which induce a relatively well-optimized set of abstraction domain operators. We described the use of such operators to replace the functionality of general-purpose, heavy-weight refinement algorithms that are widely used in TVLA. We demonstrated an analysis speed-up by an approximate factor of 20-25, for a small set of mini-benchmarks, and we verified the precision of our analysis compared to that of TVLA.

This work calls for immediate extensions, in the form of a more thorough experimental session, and in particular such that attempts to test the scalability of the analysis for bigger, complicated, real-world programs. Beyond that, further effort is required in order to apply our tool to achieve practical shape-related applications, like the compile-time GC framework described in [2]. Being able to prove the applicability of such a framework for practical purposes would be a highly desirable goal.

A separate possible effort should aim at reducing the cost that is associated with abstraction inflation, due to the high complexity of the abstraction domain. Several ideas in that direction are being examined, and hopefully will be brought to practice in the future.

## References

- [1] G. Arnold. Combining heap analyses by intersecting abstractions. Master's thesis, Tel-Aviv University, Oct. 2004.
- [2] G. Arnold, R. Manevich, M. Sagiv, and R. Sham. Intersecting heap abstractions with applications to compile-time memory management. Technical Report TR-2005-04-135520, Tel-Aviv University, Apr. 2005. Available at <http://www.cs.tau.ac.il/~rumster/TR-2005-04-135520.pdf>.
- [3] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282. ACM Press, 1979.
- [4] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.
- [5] T. W. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *ESOP*, pages 380–398, 2003.
- [6] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 24(3):217–298, 2002.

- [7] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, pages 530–545, 2004.